AUX User's Manual

Authors: BJ Kwon, JH Park

I. Introduction

AUX stands for AUditory syntaX, used to define arbitrary sound signals and operations/processing of auditory signals. It is semi-mathematical, i.e., signal components can be added, subtracted, multiplied or divided. Unlike MATLAB, two signals with different lengths can be added or multiplied without worrying about the indices or vector lengths—even the operation between a signal and a scalar is possible. Users only need to be familiar with conventions in such operations.

II. Basic format of signals and operators

2.1. Terminology

* **Vector**: An array of values

* **Signal**: Same as the vector, but connoting a "sound" signal; AUX itself doesn't differentiate a signal from vector, but the different use of these words might be helpful for this document. Conceptual difference: you conceptualize a vector as an array of values with a certain length in "sample counts;" whereas you conceptualize a sound as a sound with a duration in "seconds." By definition, all signals in AUX specifications are time-bound; therefore, a duration (or sample count) is determined when you define a signal.

* **Scalar**: A constant value, a special form of vector
* **Variable**: A variable can be used to denote or define a vector or signal and use it later.

Below, A and B are signals, a and b are scalars.

For arithmetic operations, regardless of the duration of the signals, the operation is applied as long as each operand is valid and available. For example (note that we haven't discussed how a signal can be defined in AUX, signals here are defined in verbal expressions):

A = (a 500-Hz tone from 300 ms til 600ms)

B = (white noise from 500 ms til 1000ms)

Then A+B is a 500-Hz tone from 300 ms til 500 ms, the same tone plus noise from 500 ms till 600 ms, then noise from 600 ms til 1000 ms. The same principle applies to multiplication.

For the operations between a scalar and a signal, essentially the scalar operand applies through the

entire duration of the signal. From signal A defined above,

0.2 * A is the signal A multiplied (scaled) by 0.2 for the entire duration of the signal

0.2 + A is the signal A plus a dc-shift with the DC amplitude of 0.2.

The above explanation appears more complicated than it actually is. And this kind of convention allows flexible application of arithmetic operations, particularly with the concept of "null signal (see below)." If you are unsure now, please keep reading this. By the end of section 2, you will have a clear idea about the fundamental framework of AUX. Also, in the examples below, "functions" are used without explanations, just keep reading it, those will be covered in section 3.

2.2. Time-shift operator: >>

`A >> a` → Shift the signal A in time forward by a milliseconds.

Example1) `tone(1000,100)` is a 1000-Hz tone with 100 ms. `tone(1000,100)>>500` is the same tone but begins at 500 ms instead of 0.

2.2. RMS scaling operator: @

`A @ a` → Scale the signal A at a dB rms re 0 dB (full scale of the sound card)

`A @ B @ a` → Scale the signal A at a dB rms re 0 dB (full scale of the sound card)

Example2) Let A be a waveform read from a wave file: `A = wave("bob.wav").` Then `A @ -20` specifies the signal rescaled to make its rms -20 dB re the full scale. `A @ wave("masker") @ 5` specifies that the signal be adjusted its scale to its rms 5 dB above that of another signal "masker.wav."

2.3. Extraction operator: ~

`A(a~b)` → Extract a time portion between a and b millisecond from signal A. If b<a, the extracted signal is time-reversed.

Example 3) From the signal A defined in Example 2, `A(300~500)` is the extracted portion of the waveform from 300 to 500 ms. Suppose that `A` has a duration of 1000 ms, then `A(1000~0)` is a time-reversal version of signal `A`.

2.4. Playback adjustment operator: %

The % operator changes the playback rate the signal—i.e., modulating the pitch of voice and making the duration faster or slower accordingly.
Example 4) Continuing from example 5, `A%2` is the signal of the voice with the half of the pitch (with doubled duration), and `A%.5`—or `A%(1/2)`—is the signal of the voice with doubled pitch and half of the duration.

2.5. Stereo signal

`[A; B]` → A stereo signal: A on the left channel, B on the right channel. A and B need not have the same duration, nor they may have entirely unrelated time arrangement.

2.6. Null signal or null portion

A null signal is defined as "a signal that is not defined." And null portion is the time portion that the signal is not defined. As such, arithmetic operations of a null signal (or operations during the null portion) have no effect. Note that null signal is different from a signal with zero value. For example, in the signal defined as `tone(1000,100)>>500`, the signal is not defined from 0 to 500 ms. The concept of null signal is useful to apply arithmetic operations only for partial portion of a signal.

Example 5) For ramping/damping of a signal, envelope-smoothing vector should be multiplied at the onset and offset. With the concept of null signal, this can be easily done, as follows:

```
window1 = tone(25,10)*tone(25,10)  // cosine-square function for increasing arc for 10-ms
window2 = window1(10~0)>>490  // mirror image in time for offset
window = window1 + window2  // defined during the onset and offset portions, null otherwise
// Finally, smoothing occurs only during onset and offset
smoothed_signal = window*original_signal
```

2.7. Further basics

The above example illustrates why AUX is easier than in MATLAB, because management of indices or vector lengths is not necessary but only consideration of time in millisecond. But even easier, the ramp function is provided in the AUX library to apply a raised cosine-square function to a signal for a specified ramping duration. And even further, if the user needs a different type of envelope smoothing—such as a different windowing or different times for onset and offset, etc—then a user-defined function can be prepared and used for repeated uses. In other words, the user does not need to

write the codes in Example 5 over and over again.

Finally, many MATLAB-style expressions are also valid in AUX language. This feature was added, first, to help seasoned MATLAB users transition to AUX easily, and second, to define and represent any signals or processing that can be done in MATLAB. As of now, not all MATLAB expressions are supported in AUX (probably all of them will not ever, because it is not necessary to bring the restrictions in MATLAB to AUX). Plus, if you specify a sound from scratch as a vector, you will need to consider all the low-level stuff such as the sampling rate, sample count, etc. See the following two expressions that both produce the same sound in AUX:

```
tone(440,1000)
```

```
sin(2*3.141592*440*[1/fs:1/fs:1])
```

The first expression is in the original AUX-style and the second is MATLAB-style. Do you see the difference? In AUX you don't think about the actual implementation of the sound as the digitized samples in the digital domain, rather you consider it as an abstract entity, "sound with certain properties," i.e., frequency, duration, etc. The presumption is that the issues of implementation are all taken care of by the program that uses AUX library, not by the user. Therefore, in this AUX document, the sampling rate is not considered or discussed at all (in realistic applications, the application should provide some mechanisms to adjust (or set) the sampling rate to meet the demand in hand, so that the user can set the sampling rate properly (for example, if a 15000-Hz tone is to be created, the sampling should be greater than 3000-Hz), or if the programmer wants to fix the sampling rate at one value and does not want to allow the user to change, at least it should notify the user if it exceeds the Nyquist rate). However, in MATLAB language, you construct all the digitized samples from scratch and in doing so you need to explicitly specify how digitization should occur by supplying the sampling rate every time you define a signal.

It should also be noted that MATLAB-style expressions are supported in AUX to define or construct vectors (true arrays of values, not actual sounds).

To summarize, the definition of signal in AUX is high-level, while the definition of signal in MATLAB-style is low-level. It is user's choice to use which option is proper for them. For the most part, MATLAB-style can be avoided, or at least the repeated use of MATLAB-style expressions can be avoided by using user-defined functions.

Here are a few MATLAB-style expressions or conventions supported in AUX that might be useful:

i) To define an empty vector: do this V = [ ] → Don't be confused this with null signal. We just need this definition to initialize an empty vector for subsequent accumulation. For example, V = [ ] then later V = [V something_else] in a loop or something.

ii) A vector can be defined in brackets; e.g., V = `[0 20 40 60 80].`

iii) A vector can be defined by the colon (:) operator, e.g., `V=0:10:1000`

iv) The colon operator works a little differently from in MATLAB, if there is only one colon. You figure the difference from the examples below:

`1:5` is equal to `[1 2 3 4 5]`
In AUX, `5:1` is equal to `[5 4 3 2 1]`
In MATLAB, `5:1` is equal to `[]` (empty)

In other words, if you need to define a vector `[5 4 3 2 1]` in AUX, you may specify it as `5:-1:1` as you do in MATLAB, but in addition, you may do `5:1` to make it simpler→ you will enjoy this simplification.

v) As in MATLAB, indexing is one-based.

Example 6) If x is defined as a sequence with the step of 10, from 0 to 1000, `x=0:10:1000`, then `x(1:2:9)` indicates the following vector `[0 20 40 60 80].`

Example 7) Continuing from examples 2 and 3, `A(22050:1)` is identical to `A(1000~0)` in example 3 (presuming that the sample size of `A` is 22050).

vi) Cell arrays can be defined in the same ways in MATLAB using braces.

vii) A string variable can be defined a double-quotation mark: e.g., str = "c:\soundfiles\test.wav", wave(str)

viii) String variables have the same rules and conventions as in MATLAB, such as concatenating or accessing by indices.

2.8. Other conventions (full scale, order of operations)

In AUX, we shall use 1 as the full scale. And functions, tone and noise produce the sound at the full scale (at the amplitude of 1). In the current form, AUX does not check whether the expression leads to clipping. Therefore, it is the user's responsibility to avoid clipping; e.g., `tone(440,300) + tone(880,300)` produces clipping. A proper scaling is necessary such as `0.5*tone(440,300) + 0.5*tone(880,300)`

Authors: BJ Kwon, JH Park

The following is AUX operators in the order of precedence (highest to lowest). The precedence is pretty much standard, except for a few AUX native operators which are placed between multiplication/division and addition/subtraction. Their associativity indicates in what order operators of equal precedence in an expression are applied. Personally I wouldn't memorize this, because I think we should use parentheses whenever in doubt. But I am including this information to remind you that there is a priority of operation when there are no parentheses, i.e., don't presume that the signal you defined is unambiguous to everyone (specifically, the developers of AUX).

Operator	Description	Associativity
( ) { }	Parentheses (sub-expression / function call / vector indexing) Braces (array subscript)	left-to-right
+ - !	Unary plus/minus (sign) Logical negation	right-to-left
^	Exponentiation	right-to-left
* /	Multiplication/division	left-to-right
<< >> % @	Time shift left (plus), Time shift right (minus) Play speed RMS	left-to-right
+ -	Addition/subtraction	left-to-right
:	Range	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
=	Assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Example 8) `tone(1000,100)>>500+20` is (`tone(1000,100)>>500)+20` and is an ugly super-duper clipped signal. You should write it as `tone(1000,100)>>(500+20)` if that's what you meant. But `tone(1000,100)>>500/2` is `tone(1000,100)>>(500/2),` so you should write it as (`tone(1000,100)>>500)/2` if that's what you meant.

2.9. Programming support for looping

The following syntaxes for control looping are supported, the same way as in MATLAB.

```
for… end
```

Authors: BJ Kwon, JH Park

`while … end`

`if [elseif] [else] end`

III. Signal construction functions

tone	Generate a tone signal or tonal glide
Syntax	**`tone (freq, duration, [initial_phase=0])`**
Arguments	`freq:` frequency in Hz, a two-element vector of frequencies for a tone-gilde `duration:` duration in msec `initial_phase:` between 0 (for 0) and 1 (for 2π)
Examples	`tone (226, 400)` a 226-Hz tone from 0 to 400 ms `tone (226, 400, .5)` ; 226-Hz tone of 400 ms, 180 degrees out of phase `tone ([250 500], 400)` a tone glide from 250 Hz to 500 Hz

noise	Generate a white noise signal
Syntax	**`noise (duration)`**
Arguments	`duration:` duration in msec

fm	Generate a frequency-modulated tone signal
Syntax	**`fm (freq1, freq2, mod_rate, duration, [init_phase=0])`**
Arguments	`freq1, freq2:` the boundary frequencies in Hz `mod_rate:` how many times the frequency will swing between freq1 and freq2, must be a positive value `duration:` duration in msec `init_phase:` The initial modulation phase between 0 and 1. (0 means it begins with freq1 and 1 means it begins with freq2)
xamples	`fm(500, 700, 4, 2000)` a 2 second FM tone between 500 and 700 Hz at 4 Hz

silence	Generate a silence signal
dc	Generate a dc signal
gnoise	Generate a gaussian noise signal
Notes	The same format as noise, it takes only one argument, the duration in ms.

wave	Read a wav file
Syntax	**`wave (wave_name)`**
Arguments	`wave_name:` name of wave file. Use double-quotation marks, or a string variable. If path is not included, current folder (the same directory as the psycon.exe will be searched). The extension ".wav" can be omitted.
Examples	`wave ("c:\soundData\specialnoise")` Read the signal from specialnoise.wav in the specified directory.

file	Read from text (ascii) file, the values will be read into an array, delimited by a space, CR, LF or tab character.
Syntax	**`file (file_name)`**

Authors: BJ Kwon, JH Park

Arguments	`file_name:` name of text file. Use double-quotation marks, or a string variable. If path is not included, current folder (the same directory as the psycon.exe will be searched). The extension ".txt" can be omitted.

IV. Signal modification functions

am	Amplitude-Modulate the signal
Syntax	**`am(signal, mod_rate, [mod_depth=1], [init_phase=0])`**
Arguments	`mod_rate:` AM modulation rate in Hz `mod_depth:` Modulation depth between 0 (no mod) and 1 (full mod), If skipped, full modulation is assumed. `init_phase:` The initial modulation phase between 0 and 1.
Examples	`am(noise(1000), 8, 0.5)` AM of a white noise at 8 Hz with a depth of 0.5

lpf **hpf** **bpf** **bsf**	IIR filtering. Lowpass, highpass, bandpass, and bandstop filtering, respectively. lpf and hpf require at least 2 arguments (signal and edge frequency) and bpf and bsf require at least 3 arguments (signal and two edge frequencies). The rest arguments are optional and available for tweaking detailed filter settings.
Syntax	**`lpf(signal, freq, [order=8], [kind=1], [dBpass=.5],[dBstop=-40])`** **`hpf(signal, freq, [order=8], [kind=1], [dBpass=.5],[dBstop=-40])`** **`bpf(signal, freq1, freq2, [order=8], [kind=1], [dBpass=.5], [dBstop=-40])`** **`bsf(signal, freq1, freq2, [order=8], [kind=1], [dBpass=.5], [dBstop=-40])`**
Arguments	`order:` Filter order. Default=8. `kind:` 1 for Butterworth (default), 2 for Chebyshev and 3 for Elliptic `dBpass:` Passband ripple in dB (default = 0.5 dB). `dBstop:` Stopband attenutation in dB (default = -40 dB).

Examples	`lpf(noise(1000), 2000)`
	Lowpass noise with a cutoff frequency of 2000 Hz (8th order Butterworth)
	`hpf(noise(1000), 3000, 6, 2)`
	Highpass noise with a cut of 3000 Hz, 6th order Chebyshev
	`bpf(noise(1000), 1900, 2000, 6, 3, 1, -80)`
	bandpass noise between 1900 and 2000 Hz, 6th order Elliptic with 1dB ripple and -60 dB attenuation

note	Be sure to check the IIR filter's stability. Note that some filtering specs (such as very narrow bandpass filtering) will be difficult to obtain stable filter coefficient.

filt	Filter the signal with the specified coefficients
Syntax	**`filt(signal, num_array, den_array)`** **`filt(signal, mat_file, matlab_filter_variable)`**
Arguments	`num_array, den_array:` Numerator or denominator coefficient arrays. Must use a bracket or tag (see how to use tag). `mat_file:` The matlab file (*.mat) that contains the variable specified in the 3rd argument. Must use double-quotation marks. This could include a path. Default extension is .mat. `matlab_filter_variable:` The matlab variable of filter coefficient generated by sptool. Must use double-quotation marks.
Examples	`filt(noise(1000), [.2 -0.2 .001], [1 -.2 .02])` Filter the white noise with these coefficient arrays (2nd order IIR)

	`filt(noise(1000), "filters", "lp1500")` Filter the noise with the coefficients available in matlab variable lp1500 stored in filters.mat

filtfilt Syntax	Anti-causal (zero-delay) filter the signal with the specified coefficients **`filtfilt(signal, num_array, den_array)`** **`filtfilt(signal, mat_file, matlab_filter_variable)`**

ramp Syntax Arguments Examples Note	Make the beginning and ending points smooth (to avoid spectrum splatter) **`ramp(signal, ramping_time)`** `ramping_time`: The time in ms for ramping up (at the beginning) and down (at the end) `ramp(tone(1500,500), 10)` Make the tone with 10-ms beginning and ending ramping. This uses the cosine-square function for smoothing.

V. Computation functions or functions retrieving properties of a signal

If you want to use a dB factor for scaling, the following db function comes in handy.

db
Syntax
Arguments
 	
Examples
	Calculate the actual factor from a dB value
	`db(db_value)`
	`db_value`: the scale factor of the specified value in dB
	The following expressions are the same:
	`db(-6) * tone (300,500)  == 0.5 * tone (300,500)`
	`tone (300*db(6), 500) ==  tone (600, 500)`
	`am(noise(1000), 8, db(-20)) == am(noise(1000), 8, 0.01) (-20 dB Amplitude Modulation)`

rms Syntax Arguments	Compute the rms value in dB **`rms(signal)`** `signal`: any valid signal

dur Syntax	Get the duration of signal in ms **`dur(signal)`**

rand Syntax Note	Generates a double-precision random number **`rand(max_val)`** Double-precision random number between 0 and max_val (with a uniform distribution)

irand Syntax Note	Generates an integer random number **`irand(max_val)`** Integer random number between 1 and max_val (with a uniform distribution)

randperm Syntax Note	Generates a randomly permuted array from 1 to val **`randperm(val)`** An integer array from 1 to val and its order is randomly permuted is generated.

tbegin	Get the beginning time in ms (first time point where the signal is non-null)
tend	Get the ending time in ms (last time point where the signal is non-null)
mean	Get the mean value of signal
max, min	Get the maximum (or minimum) value of the signal
Syntax	**`tbegin(signal)` `tend(signal)` `mean(signal)` `min(signal)` `max(signal)`**

Also the following functions are available for numerical calculation.

sin	Sine
cos	Cosine
log	Natural log
log10	10-base log
abs	Absolute value
exp	Exponential
^	a^b (a raised by b)
sqrt	Sqrt root
mod	Modulo

VI. Other functions

These functions were added recently. I am certain that I will keep these but I haven't yet decided how to categorize them.

left	Extract left or right channel from a stereo signal
right	
Syntax	**`left(signal)`** or **`right(signal)`**
Arguments	`signal`: stereo signal

fft	Fast Fourier Transform or Inverse Fast Fourier Transform.
ifft	This computes FFT of the signal and generates its spectrum in a two-channeled signal.
Syntax	**`fft(signal,[nSamples=512])`** or **`ifft(spectrum)`**
Arguments	`signal`: mono signal (if stereo, the second(right) channel will be ignored)
	`spectrum`: a stereo signal containing the spectrum of a real signal (i.e., its left and right signals must be Hermitian conjugate)

VII. User-defined function

Users can make their own functions—if you don't like the functions that are provided, you can override them. It needs to be saved with the extension of *.aux and accessible by the program. It has the following format (similar to MATLAB):

```
function [output1, output2,…] = function_name (input1, input2,…)
```

…

```
Main body of the function—that prepares and specifies output
variables as results
```

Similar to MATLAB, the `function` keyword declared in the top of the file is visible to the program. If there are more functions defined in the middle of the file using the same `function` keyword, these are only accessible within the file (i.e., local functions).

The access of aux files is determined by the actual application using AUX; i.e., the actual program determines the location (directories) to look for user-defined function aux files.

For further information on User-Defined Functions, go to:
http://auditorypro.com/download/aux/UDF.htm

The following functions were added to assist UDF coding but they also work outside of the UDF context. I will keep these for debugging purposes but it is still debatable whether these should be part of AUX specification or not.

fprintf
 	
Syntax
Arguments
	Opens and writes to file, the same as `fopen-fprintf-fclose` in C (except that this always opens the file in the append mode).
	`fprintf(filename,"format",var_list)`
	The syntax is equivalent to the following C codes:
	`FILE* fp = fopen(filename,"at");`
	`fprintf("format",var_list);`
	`fclose(fp);`

show Syntax Arguments	Pops up a message box and displays a string. This is useful when writing to a file is not an option. **`show("format",var_list)`** See above for fprintf